ProgrammingParadigms-Lecture27

Instructor (Jerry Cain):– cell phone off. Hey, everyone, welcome. I don't have any handouts for you today. You should pretty much be done with all the handouts. Yesterday's section solution hasn't been posted on the web yet, but Ryan will do that now. And the next time you are going to see me will be Monday morning at 8:30 or Monday afternoon at 3:30.

Remember you can take the final at either time. You don't have to tell me ahead of time which time you are planning on taking it. You can only take it once. But I will post the exam as a handout at 3:30 that afternoon, so SCPD students who are watching me right now just plan on downloading it. I am a little more flexible on when you take the exam because I know you work, but I need the exam faxed back from you, faxed in by Tuesday at 5:00 because we are gonna crank on grading that Tuesday night.

You have an assignment due tomorrow night. You can use as many late days as you want to. It was designed so that you weren't supposed to use five late days on it, but if you have five late days and you want to consume them and hand it in after the final exam that's not a problem because we are not going to be able to grade that until after the final anyway.

You should definitely get – if you haven't got Assignment 4 back yet, then let me know because you should have got those back. I have seen all the grades fly through my email. Assignment 6 is being graded right now. I told my TA's I have to have that back. It's a seashell. They have to have the Assignment 6 back to you by the weekend so you can make sure that you understand shredding to the extent that I am going to test it.

Assignment 7 I actually don't think you are going to get back by the final. But that was the Scheme assignment. Historically, if there are 200 of you in the class, 195 of you get A's or A-'s on that because you all get it working and it's this new language so we actually have – I don't want to say we have low standards, but we just don't press you on style that much.

But if you've got that working, then you are probably fine, so you are not going to be surprised by anything in terms of your feedback on that assignment. Okay, so I left you with enough Python to get the assignment done. What I did is I invited a coworker of mine from Facebook who is responsible for that noise. No, I invited a coworker of mine.

He was actually kind of my boss for the first six or seven weeks of the time I was there, and he is a programming language enthusiast as well. He knows two languages called Haskell and ML that I don't know as much about. And I asked him to speak about one of them called Haskell.

So I am just going to hand it over to him and let him kind of present a language to you that you would not have otherwise seen had I been the only one teaching the class, okay.

So let us do a little bit of a microphone switch right here. This is Sasha Rush and I will do this. And I will let him talk.

Guest Instructor (Sasha Rush):Okay, so I am gonna talk today about Haskell, which is kind of like an avant-garde programming language. It's not something you will see in a lot of companies using these days. If you want to get a job, you should be a really good C programmer. If you want to program the future you should learn a lot of Haskell. Think of it like as like the Schaumburg of programming languages.

So here is a little history of Haskell. It comes from the same family as like Wisp and Scheme. It's a functional language which means setting variables is discouraged. Using functions all the time for everything is really how you got stuff done. There is things like map that you will be used to from Scheme but there is like a million less parenthesis, so that's really good.

Then that kind of went down to these languages called ML and there is a language called Ocaml which is like a modern version of ML that is pretty popular. These languages took Scheme and kind of merged it with C in an interesting way in that they are statically typed. So you have integers that are actually integers and if you to try to like use an integer with a string it will fail.

That's really nice because you add a lot of speed to the program but it's really annoying in like C because you have to type integer everywhere. So the kind of brilliant idea behind ML is you write code that looks like Python code, but it figures out the type for you and so it will throw an error but you don't have to put any of the definitions down. It's like pretty amazing that is brilliant idea that was invented like 30 years ago isn't used today, but I think more and more it will become pretty standard.

The latest version of C Sharp and future versions of Java Script will possibly have this kind of stuff in it. So then this language came out called Miranda, which was like a proprietary language, but had this really neat idea that we will talk about a little bit later which is that it had this thing called lazy evaluation. And lazy evaluation you won't see in pretty much any language but Haskell.

The reason of that is all the researchers working on this problem got together and they said, "Hey let's make kind of an open source version of Miranda that's lazy and is standard and we can all work on it together." So, it's a really good thing if you want researchers to look at your language don't make it proprietary.

This all came together in Haskell 98 which, as per the name, came out in 2003. And that's kind of like the current modern version of Haskell. Since it came out in 2003, it's been incredibly popular with a small group of pretty dedicated people. And they have been kind of working on it pretty hard core for the last five years. It's getting to the point now where it's like really good.

Okay, so Haskell is pretty neat. So it's safe like Java. So when I say safe I mean it's really hard to make a program that compiles and then fails. So this is like really knowing about a language like Python is that you are constantly running your programs, they're constantly failing, you are constantly writing them again. It's really nice because, particularly for applications that matter, things that aren't websites that the program that you compile you know it's not going to fail for like a dumb reason that you like left something in.

So what does this mean? Well, this means if you try to add a string and an integer it's going to fail. The little colon syntax is kind of like the cons operator in Scheme. It means like put this at the start of a list, so if you try to put a string in a list with numbers it's also gonna fail. So this again is different than Scheme because Scheme is not statically typed. Scheme will let you put anything on the list.

In Haskell, if you have a list it has to all be the same kind of type of thing. Okay, so it's like Java, but safer. There a lot of kind of funny issues in Java. Here's one of them, so, I have an object called temp and then I set it to null. Then I try to run this method that I like and know it should be on object, like it's guaranteed for me to be on object, but it can fail because temp is null.

So like this is a pretty common thing that most people are used to in [inaudible] languages but it's this huge issue like there's all the possible run time errors that like you are not protecting for. Here's another one. so we have this guy named object, I'm sorry, named temp that was an object, and I am trying to task it to become a string which is okay in Java. You're allowed to – I think it's called downcast. You're allowed to move from one guy to its child.

And it has this huge possibility of failing because in this case it's not a string; you can't pretend it's a string. Okay, it's expressive like Scheme, so like I said before, so you have this map function. So the kind of square brackets are a list. It's kind of a list literal. It's similar to the list literals in Python and you can run kind of anonymous functions over it.

So this function maps adding one to a list and you get back a list with all the numbers plus one. Here's another example, this is a sort by function. Sort by takes a comparison function and does a sort with that function, so this one here just sorts a list by the standard ordering.

Okay, it's fast like C. It's not as fast as C but it's getting there. It's kind of like on the order of magnitude and here are like some stats. This is not the way to think about speed of languages but it gives us an easy way to look at it. So this is from the computer language benchmark game online. Just to give you an idea, the big difference is that types not only guarantee like safety.

They also allow you to make the language pretty fast. You know that when you see a plus sign that you are going to be adding like two integers and not going to have to check for this failure case, and that makes the language a lot faster.

Okay, most importantly, it's pretty fun. When I say fun, I mean like if you want to do a multiplication function you just do a multiplication function. You don't have to like write all other stuff around there. You can do things like mixed arrays. This is kind of like the tuple syntax in Python, I don't know if you have seen that in this class.

And finally, it's got a lot of these like crazy things you can kind of just throw in. Jerry said that you guys saw very briefly list comprehensions in Python, so list comprehensions were idea taken from Haskell. This is a syntax for taking all the numbers from one to ten, multiplying by a two and giving that back as a list. So, this guy on the right here is kind of like range in Python and the whole thing is like a map.

Okay, so I should say now that I am going to show a lot of code examples as we go through Haskell, so feel free to interrupt me at anytime and ask questions. I wasn't really exactly sure at what level I would be doing this at, so some of this maybe like a little bit crazy.

Okay, so we are going to look at Fibonacci, which is like pretty standard, I guess, programming example. So, for you guys who don't remember it, the Fibonacci sequence is simply the sequence where you take the last two numbers, you add them together and you keep on going and going and going and going. So, this is kind of a definition.

Okay, so, here's the Fibonacci sequence in Java. So we have a function fib. We give it an N, which is how long we went the sequence to be. It returns a list of integers. Java is really nice. It has this new generic syntax that lets you say what the list is of. So the first thing we do is we declare. We declare the list. That's our sequence right there, we set the first value, we set the second value, and we do a for loop.

It's pretty simple. It kind of describes the program that we are doing step by step, each line says what it is doing, does something, and moves on to the next line. So I think that's pretty good. So this isn't in Haskell. So where did it all go? Where's all the coding, right? Where's all the writing?

So in Haskell we take a different approach. We go back to the definition of what this thing was and we like just write it, okay. We are not concerned with how the machine represents it, we're not concerned with the ordering, how it's presented to the user, we are just concerned with, like, what's the math? Like, what's going on here?

And you can kind of describe it in the most basic of ways. And so I told you that this is safe, I told you that this is fast. So you can see that it's fun but like the other two, it's hard to describe what is actually going on here that makes this work. So, let's dive down into the details.

Okay, so, here are the two bodies of these functions. So let's try to map like each part to each part. So I said before that the colon means a list operator. It means like hey, put this at the start of the list. So we can see that we started off with the one in our list. So that maps down to hey, this sequence of zero is equal to one. Then we can see the second part is equal to one. So that part, I think, makes sense.

The part I am going to have to sell you on is what this last guy does. So, what's going on here? So one thing that you can see is that we are using the word fib within that expression and we are also setting fib. So it seems really weird that you can use the variable that you are setting within the variable itself. So that's a little bit strange to get over.

So the reason that this works is that we want to think of this not like a variable but almost like a function. So it's like a recursive function that you have probably seen before. So if we think of fib as a function it makes a little bit more sense, we are taking like the old value of fib and putting within the use of it. What does tail mean? Tail is similar to, I guess, like [inaudible] in Scheme or tail in Scheme. Yes, CDR it means take the end of a list not the first owner.

And what is this zip with doing? Zip with is kind of like the map function. It says, "Hey, map this function over this list." Except instead of just mapping the function over one list it takes two lists and combines them with the function. Okay, so, we take the two lists, we combine them and we are good. So, let's look at this in more detail. So, first of all, how can we use fib as a variable?

Well, what do we know about fib? We know that fib starts with one-one. It's a list that starts with those two values. And after one-one it's a question mark. We don't know anything about what's going on after that. Okay. What do we know about tail fib? Well, we know it starts with one and then we don't know anything about it. Right? It's the same blankness that we saw before.

So then the question is what is the zip with guy? So as I said before it's going to add these two guys together. So we know we have won one and we know we have the tail, so we zip them together and we get two. Once we know that's two that goes to the end of the other two lists. So we have it there. We then add these together again, we get three and we keep on going.

So, what happened there? So the trick is that Haskell is what's called a lazily evaluated language. And that means it will do no work at all until it has to. It's like the undergrad of languages. So, we start with one-one and then we go through like zip with. It's all in what's like this blocked. It's like in a jail. It's called like a funk.

It's something that's there that we have to evaluate that we haven't evaluated yet. So then I say, "Hey, I want to take the first five values of this guy and I want to get them out." So then what happens is the first five values get kind of like pushed through, they get converted from like funk land to like re4al values. And then we just leave the rest. We just say, "Hey, we are going to evaluate that sometime in the future."

And that’s pretty cool. It’s cool in that it means that we can kind of have infinite lists. This list can go on forever and it’s totally cool in your program that the list is there because you are not going to spend infinite time evaluating the entire list all the way through. A good comparison like this is the X range function in Python.

So instead of actually constructing the range of the entire list it simply just takes one value at a time as you request them. So it’s kind of like that except for everything, so that’s like that’s the default standard. So, here’s a good example, well let’s see, let’s skip this guy.

So, one thing that’s kind of frustrating about programming languages is that if you want to have a function that kind of works like the if statement does it’s really hard to write it. So imagine we have a function in Python called my it. And you take three values, say like Boolean value of like what branch to go to and the if branch and the out branch.

And then if the first guy is true we take – oh, sorry this is a typo. If the first guy is true then we take the B branch. If the second guy is true – if the first guy is false, we take the C branch. Sorry, totally out there. So, why can’t we do this in Python? Anyone have any ideas? So, what happens if the B value is like print hello? So, if you have any kind of like effects in either your then branch or your else branch, like print, for instance, or like, I don’t know, you make like an internet request, then you have this problem where if that branch doesn’t get called that value still goes out on the screen.

Okay, so if you say like if – if true print hello, if false print goodbye, right. You’re gonna get hello goodbye because both those statements are gonna get evaluated before they are sent to the function, and then the function is just going to return the results, okay.

So in a language that’s lazy nothing gets evaluated, it just remains in its funk until it’s needed. And then when it’s needed, it gets evaluated. So the branch that failed would just go away. Any questions with that? Okay.

So then the second question is well how can it be fast? I said before that to be fast you really need to know the types of what’s going on. And if you look at our old guy we didn’t write any types at all. There are like no types. There’s no ints, there’s no functions, there’s no lists, there’s no anything. So the trick is that Haskell, it kind of like figures out the types.

So I am not going to go into this too much. There’s like a whole body of study on how to do this correctly, but we can kind of just play with it. So, what types do we know to begin with in this function? Anyone? What values here have types? Yeah, the one and the one; we know these guys are both ints.

Okay, and what’s the type of fib, what did I tell you about lists that we know about the type of fib? What’s that?

Student: Homogeneous.

Guest Instructor (Sasha Rush):Homogenous, right. Okay, so if the ints are both there, okay. I'll stop asking questions, but if the ints are both – if we know that they're both ints and we know they are in a list, that means we know that fib must be a list of ints, right? Off hand, even to start with? Okay, so if fib is a list of ints it means this last guy has to be a list of ints or we will fail, okay. So let's check that it is.

Well, we know that fib is a list of ints because we just said so and we know that tail fib also must be a list of ints because it's just the tail of a list of ints, and we know that zip takes two lists and adds them together, so therefore it's also going to be a list of ints. So that means the whole thing type checks and we're good. So the complier knows what the types are, it can make it faster. We know that we didn't make a mistake, so it's safe and everyone is happy.

Okay, cool. So now let's go kind of more deeply into what the types are within Haskell and look at them. Okay, so we are gonna start off with a bunch of basic types. We're gonna have an integer type, we're gonna have a float type, we're gonna have a character type.

Okay, and like Scheme, functions themselves are also gonna be a type so we can pass functions around in that way. So the way we look at types of functions is with the kind of double colon notation. Think of the double colon notation like a declaration in Java that just says this is what the function is. So, what did it say?

Well, we have a function called add one. It takes a number and it adds one to that number. You don't need to use the word return at all in Haskell, so the result it's returning is the number plus one. So the way that we write it is we write int arrow int. That means we take an int and we return an int.

Okay, how did you do adding up two numbers? You write the word add, you write the arguments as spaces after the function. So it is add, val one, val two and we just add those guys together. Now this guy is a little bit confusing, the way you write the type signatures when you have two arguments is arg one, arg two, return value, okay. So that's just how we write functions.

Okay, so there's another kind of type and this is the type like the user defines. So in Haskell you are allowed to define your on types and kind of just put them into the system. Here is the data type for Boolean. So a Boolean type can be either true or it can be false, okay. We put that guy in there. It's now a type in the system. We can write the word true it means something. We can write the word false it means something.

Here's how characters are represented. Characters is just an or with all the different letters possible. Here's color, so color can just be like red, green, blue, whatever. You can kind of write any of these that you want, just like anything you make up you can just put into the language itself.

Okay, so, now we're gonna look at how you do a function over this type. So this is the not function. It takes a Bool, produces a Bool. Bool is this new type we just added to the [inaudible]. So here's the function, so it takes a val one. If the val one is equal to true, so we're now using our type true then we are gonna return false, also we're gonna return true. Okay, pretty simple, these are just new things we have added to the language.

But what's even cooler is that you can use your types in a different way which is called pattern matching. The way pattern matching works is instead of doing before I can write the same exact thing and the same thing we just wrote as this. So this means the not function when given the pattern true will return false, when given the pattern false will return true. So I matched on the type that I want and I have a specific value for that type. Yeah.

Student: When you define a function do you always need to both give the typed information like the Bool arrow Bool and also the definition?

Guest Instructor (Sasha Rush):You don't actually have to do it. It will like figure it out for itself. It kind of uses documentation though, so it's like a good way of just telling other people that read your code, "This is what the function does itself."

Also, people write them sometimes because sometimes Haskell will figure out the type but it will be very generic so it will kind of generalize your functions for you and sometimes that can be a little bit confusing when you get error messages and things like that. So it's often good practice to kind of put that there.

Okay, so let's look at pattern matching a little bit more in depth. Here's the and function. This is kind of like the Boolean combinatory and. It takes a Bool, takes another Bool, and produces a Bool. So if one is true and one is false, it returns false. If one is false and one is false, it returns false. If one is false and one is true, it returns false. And if one is true and one is true, it returns false, okay.

Okay, so if you've programmed in C and newer versions of Java, you maybe familiar with this concept of an enum. An enum is often used in a similar way if you want to do true/false or colors or something like that where each one of these statements is assigned an integer value. So false maybe assigned the value zero and true maybe assigned the value one.

That's fine, but it's pretty much like a huge hack. Like it's really confusing because some things you can add different kinds of enums and things like that. So just to show you that these are not the same thing, so here is our apple type, here is our orange type. And if we try to take two values of these guys and compare them, it's going to fail. It's going to say, "Hey, these are not the same type." Just like when you tried to add a string to an integer, it's not allowed, that's a fail, we're done.

Okay, so now we're gonna talk a little more about user types. The other thing that is kind of cool about these types is that you can parameterize them with other values. Yeah, I'm sorry. So here’s the orange type. So now we have this other orange and – well, so first of all, we have seeds and seeds has two values. You can be seeded or you can be seedless. So again, just similar to like true/false.

And then we have orange which looks the same as before except we have this other guy that has a kind of space and then another type put into it. So you can think of that guy as like an argument to the first type. So it’s just when you use this in practice you can say, “Hey, I have my orange and my orange is a navel orange with seeds or it’s a navel orange without seeds.”

You can kind of put arbitrary other types as like arguments to the types originally. So this is kind of similar to a constructor in an object or a new language where you kind of make a new instance of the object you can kind of pass an argument that kind of are properties on the object.

Okay, so I said before that in Java any object can be just null [inaudible]. So I kind of discouraged that before because it can be really dangerous if you don’t know an object is going to be null, it’s kind of bad and you can cause failures. It also really useful in some cases because sometimes things just fail and you want a null coming back as kind of a marker as that hey, this guy doesn’t exist, it can’t exist, it was a failure.

So we can represent this guy in Haskell too. So here’s what string, we are going to add a new type to our language called string stuff, and it can have two values. So the first value is that is can be null, and the other value is that it can have some stuff in it, and that some stuff pertains to the string that it’s supposed to be, okay.

So it’s just like a string in Java in that it can at any time be null or be some value and it has that valued contained within the type. So like this pretty good. It will work for strings, but let’s say that I have some new data type like oranges and I want oranges also to possibly be a null or have some [inaudible] that we want to look at.

The problem is that this guy only works for strings, so I am going to have to write an orange stuff too. It looks pretty much exactly the same, but works for oranges. So what we really want to do is parameterize this definition here, not only with this value which is the argument base, but also with the type of that argument so that we can maintain our type safety without breaking things.

So is this kind of like lists in Java. So instead now we’re gonna have a data type called stuff. So now I have this argument on the left side too which is like the type of the sum guy. So it’s just like an argument to a function, but an argument to a type definition, and we can kind of carry that argument through. So now if I want a string I just type stuff string and then the some can contain a string element and so forth. Yeah. Student:

[Inaudible].

Guest Instructor (Sasha Rush):So you’re talking about in this null?

Student: Yeah.

Guest Instructor (Sasha Rush):So the type of null would still be STR stuff. Let's see. So null since it's defined in the data as STR stuff, its type is STR stuff.

Student: So it will be reserved for STR stuff?

Guest Instructor (Sasha Rush):Yeah, sorry. You can't actually use it in both. Yeah, yeah, definitely, yeah, yeah. If I tried to type in both definitions it would just fail. And this works with any type A. Okay, so here's an example. So we're gonna try to do division, but we are going to make it safe so that if you try to divide by zero, we'd fail. So, if you divide by zero we return null. Oh, sorry, this shouldn't say object. It should say stuff.

So, when we try to divide by regular value, we return some, and then that value itself, okay. Why can't we just do this guy for the second part? So why do we have to have the word sum there? What's that?

Student:[Inaudible].

Guest Instructor (Sasha Rush):Yeah, well, so we definitely know the null one, but what would happen if we did this bottom guy? Why would it fail?

Student:[Inaudible].

Guest Instructor (Sasha Rush):Yeah, exactly. It would be returning an end type not like a stuff end type, which is the difference. Cool. Okay, so let's go onto to some more kind of cool types. So now let's do a list type. So a list type is similarly typed to what you have seen in Scheme. It has kind of a pair, like a cons to start with and then it has like an end value when you get to the end, and this is also abbreviated as we showed earlier in the talk as the colon operator.

So we say that a list is some value and then this list of that value, and when you get to the end you kind of have like a blank – like a stop type. This guy is a little bit crazier than what we have seen before because it's called a recursive type definition. You will notice that I am using the same definition that I used in the data side on the actual definition and that just lets us say like, "Hey, this value contains another value with inside of it."

So like a list when you do the tail operator contains another list right there. Okay, so here are some examples. This type string defined in Haskell is just a list of chars so everything is still consistent with this type definition we showed before. So, yeah, now why this is fail? Why do we have to have homogenous lists?

Student:[Inaudible].

Guest Instructor (Sasha Rush):Well, I know why we do it, but just because it's fast does not mean we should prevent people just because. So, let's look at this definition. So what was that type of that thing we just showed? So this guy is type list char because they're all characters. The guy at the top is a type list int because they are all integers. So when we try to do this guy we have integers and characters kind of intermingled in the type definition.

So we look back at this guy. Well, we said a list starts with a type, so if this guy is type int then we can only have type int inside of it, right? So we try to put strings inside of it or chars inside those two aren't going to match. That definition is gonna fail and the compiler is gonna throw an error. Okay, so there's nothing special here, nothing like magic keeping the list consistent, it's just based on what the type of the list is.

Student:[Inaudible].

Guest Instructor (Sasha Rush):So there are kind of like ways to do it, but you lose again like speed and safety. We'll show you one a little bit later that's one possible way of going about it, but it's kind of like [inaudible], how to do that well. Okay, so let's look at some kind of functions over these kind of objects.

Okay, so here like sum. So what sum does is it takes a list of integers and it returns a different integer. So again we're doing pattern matching just like we did before with true and false, but here the pattern is hey, is this a blank list. If it's a blank list, then we return zero. And then the other thing we're pattern matching on is just like we did before with pattern matching an int and a list of ints right in the function, okay.

We take those values, so we have a value next and a value rest, and we can kind of act on them just like they're values. So all we're doing here is adding next to the recursive sum of the rest of the list. Okay, here's the next guy. This is map. You have probably seen this definition before from Scheme.

So what is map? So this is the first time we have seen a function used as a type within another function. So as I said before, the way we read this guy is map is a function that takes a function from sum type A to sum type B. Then it takes a list A and it returns a list B, okay. So a map over an int list would take a function from ints to ints, take a list of ints and return a list of ints, okay?

So what do we do? Well, if a list is blank, we do nothing. If the list has some elements, we just apply the function to the first element, and then we put it at the start of another list that's the rest of the elements with that function. Yeah.

Student: Can you add [inaudible]? Would it compile?

Guest Instructor (Sasha Rush):Yeah, it would compile. You'd get – you actually would be totally fine in these cases because these two are different types. So think of this one like true and the second one like false, so you can have them in any order because it’s just matching is this type. Yeah.

I guess I should say one other thing which is that you can also write map fun and just give a variable so is they said map fun list or LS or something, then it wouldn’t work, so switch the two because the first one would match an empty list also. Does that make sense? So it matches the most general possible.

Okay, so this is my – let’s see if we can get some audience participation for this one. So here are some types that are a little bit more crazy. So if you got – this is like – I think this pretty interesting, if you can like look at them and just take a guess of what this actually is, like what are we trying to describe here? I think we have plenty of time, so we can probably go through these.

Okay, so it can be two things. The second thing is pretty boring. The second thing looks just like our sum from before, right? It’s just has one value in it, right? So, what’s going on with the first thing? The first guy is a little bit crazy.

You can have one value and then it can have [inaudible]. It’s a binary trick, right. So the first guy is like a branch and it has like two different paths that you can go down, and the second guy is like a leaf, which is just the value itself. Okay, so let’s look at this guy.

One of the things that’s kind of cool about type definitions is that a lot of times you can just look at them and say, “Oh, I know what that does just in the type definition.” That’s why I said before they are used as documentation. They are also how people like to search for new functions on the web. They’re like, “Hey, is there a function that does this? Let me type in a type definition.”

So what does this guy do? Well, it takes a function that takes an A and a B and produces a C. Then it takes a list of A’s and a list of B’s and produces a list of C’s. So the hint is we saw this guy earlier. Yeah, this is zip with. This just takes the function and applies it to the two lists one at a time, and produces the final list.

Okay, one more. So this last guy – yeah. So this guy looks very similar to a definition we saw earlier which is of a list itself, which is a pair and then a N, except that it’s got this craziness where it has two variables it’s taking to start with, okay. So we’re used to just seeing a list parameterized by one type.

This guy is parameterized by two types and you see this craziness where it flips the types when it goes to a [inaudible]. So this guy is like a list where the types alternate. So like the first one if you give it an int and a string, the first would be an int then a string and then a string, kind of alternates as it goes. Yeah.

Student: What is the capital A?

Guest Instructor (Sasha Rush):Capital A was just like me hiding what that would be. So in real practice you would give that a descriptive name like cons or something. Okay, so here's kind of just like how we would actually write these guys in practice. The first is a binary tree. The second is the function zip with. And the last is this kind of switch list guy, which isn't actually practical for anything, but is a good way to understand how lists with mixed elements work.

Okay, so you probably learned that [inaudible] to code is pretty good and we want to use it. So in Haskell there isn't really any object referring to stuff at all. Part of the reason of this is like functional programming [inaudible] because they think it's just like function code, so there's a whole history of those two movements.

The more practical reason is that it would be kind of hard to use it in Haskell. For instance, because you can't change any values, you don't have variables in the same way. It is kind of useless because there aren't any setters. You can't change an objects value. And because it's all garbage collected there is no such thing as a destructor, so all of that stuff doesn't exist. So some of the benefits I've been referring just aren't apparent, but there is some stuff that is really good. Let's kind of try to map these concepts back to what you are used to seeing.

So the main idea of a class in Haskell is this data type. So you make a data type for everything that you want, that you would make a class for, so those two are kind of similar. Then you can think of the actual use of the data type, like the use of true or the construction of a class of a list as kind of like being an object. So it's an object of that class and you can do stuff with it.

And then finally, we are going to look at this thing called type classes which is kind of poorly named but these things kind of act like interfaces in kind of a magic way. Okay, so here is an interface in Java. I forgot what a Bool looks like, but the idea is that anything that implements this interface must have a function called equals and a function called not equals. And then – sorry, a method called equals and not equals. And then we can always call those methods on any objects of that type.

So Haskell has a similar idea called a type class. So you make a class and you say, "Hey, this is a class called PQ and it applies to type A." And if this class exists that type A, there must exist these two functions, one function is an equals function that takes the type A, another type A and returns a Bool. And one is this not equals function which takes type A, type A, and returns type Bool.

The kind of cool thing that you can do in type classes that you can't do really do in interface is actually implement one in terms of the other. So the fault implementation of not equals is simply the not operator applied to equals. So now here's like an implementation of that.

So we say the instance of the equals class for the type Bool that we defined earlier means that if they're both true then that returns true, if they are both false, then that returns true, and if they're anything else then we return false, okay.

And what does this mean? Well, this means that anywhere in your program, anywhere you want, we can now use equals equals on Boolean guys and it will just work. It will just magically work. So here's another example that is a little more interesting.

So we defined this stuff class earlier and what we can do is we can say that hey, if you have a type that already has an equals, we've already defined the class equals for it like Boolean, and you have the stuff of that guy, then stuff is automatically defined with equals too by this definition. You can kind of do this for anything. You can say two trees are equal if this definition holds. You can say like two whatever's are equals.

Equal is like a pretty small example, but what's neat is there are these kind of general type classes that work over lots of things. For instance, a lot of objects in Haskell, the map operator will work for them too. So you can kind of imagine a map operator that works on trees that goes through each element, applies a function, and produces a new tree with those elements in it.

And things start to get really powerful when you can kind of just assign this arbitrary operator, have it work. You don't even have to know what is being passed in because you know as long as it defines map your function will work on it. Okay, so I think that's about it for my actual technical definitions. I said earlier though that you can't get a job with Haskell and that kind of made me sad because I know everyone wants a job.

So I thought it would be good to kind of like go through a couple of the interview questions that we give to people and show them how it would get done in Haskell. I found that people come in and they try to write interview stuff in C and they get so confused over like hey, did I forget this number here, or should this be less than or less than equal, and it's just pretty bad.

So I think one of the things that Haskell does is it treats you to think about the problem from kind of a mathematical definition and not get worried over the details of implementation. So how to get a job with Haskell.

Okay, so here is a common interview question. I figured I would ask a couple from Google and a couple from Facebook so that way we're equally screwed if you guys try to come in and interview. So this one is you are given two assorted numbers produced an assorted list of all the numbers, so it's kind of like merging the two guys together.

So the problem here is actually not really much of a problem at all. It becomes tricky when you do it in a language where you have to kind of create a new array that has the size of both of them, kind of like fill them in one at a time, and then return that array. In Haskell though, it's just a matter of taking both arrays, seeing which element of the two at the beginning is less, merging that into a new array and kind of continuing to go along.

So let's look at the code. So the first thing, as I have said before, you always do is write down your definition of what's going on here. So we have a list of integers, another list of integers, and we're producing the merged list of integers between them. Okay, so here's the code – if both lists are blank, we want to return blank. That makes sense, right? Nothing to do here.

If one list is blank and the other guy still has values, well, you definitely just want that guy because he still has values left, the other guy doesn't, and vice versa. So that's the first three lines. They're just boilerplate. We're just putting them in to do stuff.

Now the actual interesting part is the last case where we are actually doing the merge. So you will notice here that we can take the head of both lists, write in the function definition. We say if A is less than B, well, then we want to put A at the front of our list and the rest is just, hey, merge these other two lists that we created that we had already.

And then, hey, if B is less, then we take B, that guy starts the list, and then we take the rest of them and just merge them together. There's not really any magic going on here. We're not using any laziness. We're not really even using our type inference. But it's fast, it's simple, and there we go. So there is variables where you call them whatever. It just means the tail of the list. So A is the head of the list, and AS is the tail of the list. Yeah.

Student: Do the interviewers care which language the interviewees chose?

Guest Instructor (Sasha Rush):Only if they're jerks. Student:

[Inaudible].

Guest Instructor (Sasha Rush):Yeah.

Student: Is the less than sign the only thing in there that actually forces it to infer that they're pink lists as opposed to just generic lists?

Guest Instructor (Sasha Rush):Yeah, and in fact, if we hadn't have given this type definition in there, what would it infer?

Student: Just rely on probably using the less than.

Guest Instructor (Sasha Rush):Well, yeah, just using less than sign. So in actual practice we showed the EQ type class. We'd also have a comp type class. The comp type class would have less than, less than, greater than, all of those in there. So it would infer that the type would have to be something that has an instance of the comp class, so anything that would have that.

Okay, so here's another one, this is a function A2I. This one is a little bit trickier. So what is it? Well, it's a string, which as we said before, is just a list of charts. And we are mapping in to an int so it means convert an ASCII string to an integer, okay?

So here's the code. So again, four lines, they are not that long. The first thing we have to do is define this helper function which we are calling D to I, which means convert a digit to an integer. The common trick for doing this in C is simply to minus by the zero character. Does that make sense? Since ASCII characters are in order you can just minus by zero and get us back the digit.

Anyone know why we can't do that in Haskell? I mean characters are just represented by numbers, right, we should just be able to do that, right? Well, again as we said before, since there is type safety we want to distinguish the characters from numbers so in case you accidentally add them together by mistake, it's bad news.

So we have this function ord which tells us what the ASCII value is, and we apply that and do the subtraction. Okay, then this guy is really like the brunt of what we're doing. There's a little trick here called an accumulator. I don't know if you have seen that in Scheme. It's like the way to write recursive functions when you want it like kind of – it's kind of a more powerful way to write recursive functions.

So what are we actually doing? Well, for each character we are multiplying the previous stuff that we've seen by ten and we are just adding on what that digit conversion was. So we go through, so if our numbers are one, two, three we first get the digit one. Then we multiply that by ten and we get digit two, and we add that on multiply it by ten, get digit three, add it on, and then we're good.

Okay, we're running out of time. I have one more example, and then I guess you want to finish up. Okay, so, here's a function to check if a string is like a prefix of another string. So the type definition is list of chars, list of chars to a Bool.

Okay, so what are we gonna do? Well, we are assuming that our first guy is the prefix and the second guy is the string we are testing again. So each time we just pull off the first character. If the character is the same, if they're equal, then we keep on going. We do the rest of the list. If they're not equal, we've failed because it's definitely not a prefix so we return false.

And then we just handle the two other cases. The cases are, hey, your prefix is blank. Well, that means we got to the end, so if we got to the end then it's definitely a prefix, so that's true. The other case is that the string we're testing against is false is blank, and that means the prefix was longer than the original string, so bad news, so that's false.

So that one's three lines. It's pretty simple. Cool, so anyway, I hope that this at least opened your mind to there being really interesting other languages out there. Haskell is pretty cool because it's both a research language and it's becoming more and more a practical language. There’s a really great wiki, Haskell.org. It will teach you all kinds of other crazy magic in Haskell.

The particularly big one is I haven’t showed you how to print yet, and as I hinted at earlier, it’s kind of difficult to print when it’s lazy, so there’s are kind of really cool tricks for that. There are a bunch of different implementations. GHC is the main one. You can download that for your Mac or whatever. And there’s a bunch of big programs being developed in Haskell like right now. So anyways, thanks for having me.

Instructor (Jerry Cain):That is all. I will see you all Monday morning. Don’t freak about the exam. I’m not saying it’s going to be easy, but it won’t kill you either. Okay, so I’ll see you all then. Get back to you midweek with your finals. Bye bye.

[End of Audio]

Duration: 58 minutes